jn2004.04.30

5. Context free grammars (CFG) and languages (CFL)

Goals of this chapter: CFGs and CFLs as models of computation that define the syntax of hierarchical formal notations as used in programming or markup languages. Recursion is the essential feature that distinguish CFGs and CFLs from FAs and regular languages. Properties, strengths and weaknesses of CFLs. Equivalence of CFGs and NPDAs. Non-equivalence of deterministic and non-deterministic PDAs. Parsing. Context sensitive grammars CSG.

5.1 Context free grammars and languages (CFG, CFL)

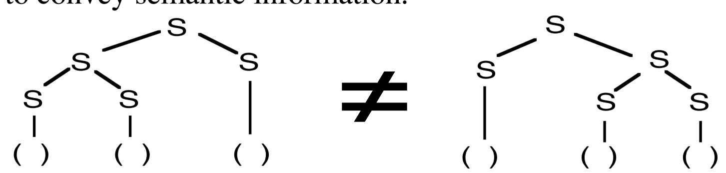
Algol 60 pioneered CFGs and CFLs to define the syntax of programming languages (Backus-Naur Form).
Ex: arithmetic expression E, term T, factor F, primary P, a-op A = {+, -}, m-op M = {•, /}, exp-op = ^.
E -> T | EAT | AT, T -> F | TMF, F -> P | F^P,
P -> unsigned number | variable | function designator | (E) [Notice the recursion: E ->* (E)]

Ex Recursive data structures and their traversals:
Binary tree T, leaf L, node N: T -> L | N T T (prefix) or T -> L | T N T (infix) or T -> L | T T N (suffix).
These definitions can be turned directly into recursive traversal procedures, e.g:
procedure traverse (p: ptr); begin if p $\neq$ nil then begin visit(p); traverse(p.left); traverse(p.right); end; end;

Df CFG: G = (V, A, P, S)
V: non-terminal symbols, "variables"; A: terminal symbols; S $\in$ V: start symbol, "sentence";
P: set of productions or **rewriting rules** of the form X -> w, where X $\in$ V, w $\in$ (V $\cup$ A)*
Rewriting step: for u, v, x, y, y', z $\in$ (V $\cup$ A)*: u -> v iff u = xyz, v = xy'z and y -> y' $\in$ P.
Derivation: "->*" is the transitive, reflexive closure of "->", i.e.
u ->* v iff $\exists$ $w_0, w_1, \ldots, w_k$ with $k \geq 0$ and u = w_0, w_{j-1} -> w_j, w_k = v.
L(G) context free language generated by G: L(G) = {w $\in$ A* | S ->* w }.

Ex Symmetric structures: L = { $0^n 1^n$ | $n \geq 0$ }, or even palindromes L_0 = { w $w^{reversed}$ | w $\in$ {0, 1}* }
G(L) = ({S}, {0, 1}, { S -> 0S1, S -> ε}, S); G(L_0) = ({S}, {0, 1}, { S -> 0S0, S -> 1S1, S -> ε}, S)
Palindromes (length even or odd): L_1 = {w | w = $w^{reversed}$ }. G(L_1): add the rules: S -> 0, S -> 1 to G(L_0).

Ex Parenthesis expressions: V = {S}, T = { (,), [,] }, P = { S -> ε, S -> (S), S -> [S], S -> SS }
Sample derivation: S -> SS -> SSS ->* ()[S][ ] -> ()[SS][ ] ->* ()[()[ ]][ ]
The rule S -> SS makes this grammar **ambiguous**. Ambiguity is undesirable in practice, since the **syntactic structure** is generally used to convey semantic information.



Ex Ambiguous structures in natural languages:
"Time flies like an arrow" vs. "Fruit flies like a banana".
"Der Gefangene floh" vs. "Der gefangene Floh".

Bad news: There exist CFLs that are **inherently ambiguous**, i.e. every grammar for them is ambiguous (see Exercise). Moreover, the problem of deciding whether a given CFG G is ambiguous or not, is **undecidable**.
Good news: For practical purposes it is **easy to design unambiguous CFG's.**

Exercise:
a) For the Algol 60 grammar G (simple arithmetic expressions) above, explain the purpose of the rule E -> AT and show examples of its use. Prove or disprove: G is unambiguous.
b) Construct an unambiguous grammar for the language of parenthesis expressions above.
c) The ambiguity of the "dangling else". Several programming languages (e.g. Pascal) assign to nested if-then[-else] statements an ambiguous structure. It is then left to the semantics of the language to disambiguate.
Let E denote Boolean expression, S statement, and consider the 2 rules:

S -> **if** E **then** S, and S -> **if** E **then** S **else** S. Discuss the trouble with this grammar, and fix it.
d) Give a CFG for $L = \{ 0^i \ 1^j \ 2^k \mid i = j \text{ or } j = k \}$. Try to prove: L is inherently ambiguous.

5.2 Equivalence of CFGs and NPDAs

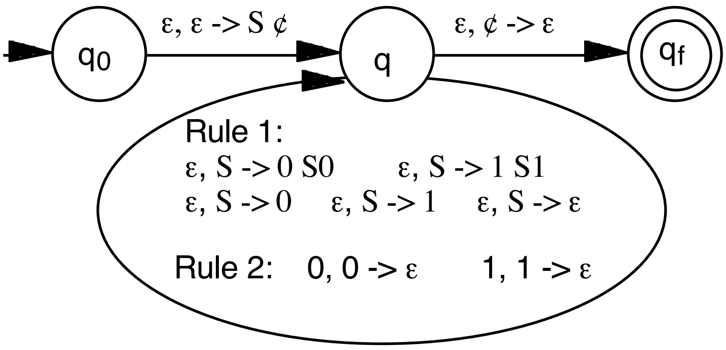
Thm (CFG ~ NPDA): **$L \subseteq A^*$ is CF iff $\exists$ NPDA M that accepts L.**

Pf ->: Given CFL L, consider any grammar G(L) for L. Construct NPDA M that simulates all possible derivations of G. M is essentially a single-state FSM, with a state q that applies one of G's rules at a time. The start state q0 initializes the stack with the content S ¢, where S is the start symbol of G, and ¢ is the bottom of stack symbol. This initial stack content means that M aims to read an input that is an instance of S. In general, the current stack content is a sequence of symbols that represent tasks to be accomplished in the characteristic LIFO order (last-in first-out). The task on top of the stack, say a non-terminal X, calls for the next characters of the imput string to be an instance of X. When these characters have been read and verified to be an instance of X, X is popped from the stack, and the new task on top of the stack is started. When ¢ is on top of the stack, i.e. the stack is empty, all tasks generated by the first instance of S have been successfully met, i.e. the input string read so far is an instance of S. M moves to the accept state and stops.

The following transitions lead from q to q:
1) ε, X-> w for each rule X -> w. When X is on top of the stack, replace X by a right-hand side for X.
2) a, a -> ε for each $a \in A$. When terminal a is read as input and a is also on top of the stack, pop the stack.

Rule 1 reflects the following fact: one way to meet the task of finding an instance of X as a prefix of the input string not yet read, is to solve all the tasks, in the correct order, present in the right-hand side w of the production X -> w. M can be considered to be a non-deterministic parser for G. A formal proof that M accepts precisely L can be done by induction on the length of the derivation of any $w \in L$. QED

Ex L = palindromes: G(L) = ({S}, {0, 1}, { S -> 0S0, S -> 1S1, S -> 0, S -> 1, S -> ε}, S)



When q, q' are shown explicitly, the transition:
(q, a, b) -> (q', v), $v \in B^*$
is abbreviated as: a, b -> v

Pf <- (sketch): Given NPDA M, construct CFG G that generates L(M).
For simplicity's sake, transform M to have the following features: 1) a single accept state, 2) empty stack before accepting, and 3) each transition either pushes a single symbol, or pops a single symbol, but not both.

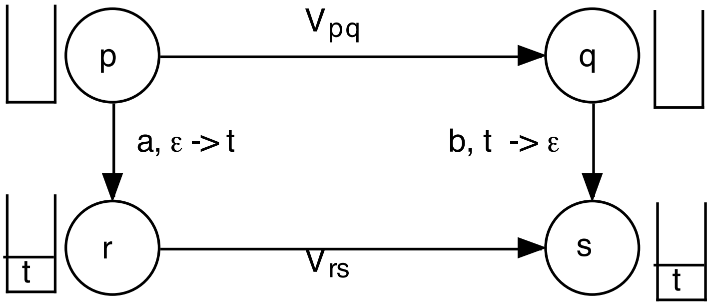
For each pair of states $p, q \in Q$, introduce non-terminal V_{pq}. $L(V_{pq}) = \{ w \mid V_{pq} \to^* w \}$ will be the language of all strings that that can be derived from V_{pq} according to the productions of the grammar G to be constructed. In particular, $L(V_{sf}) = L(M)$, where s is the starting state and f the accepting state of M.

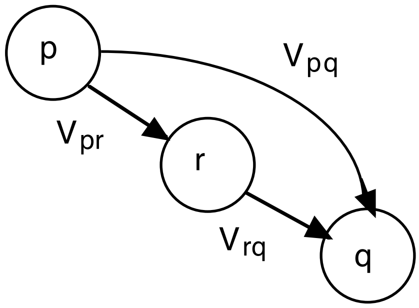
Invariant:
V_{pq} generates all strings w that take M from p with an empty stack to q with an empty stack.
The idea is to relate all V_{pq} to each other in a way that reflects how labeled paths and subpaths through M's state space relate to each other. LIFO stack access implies: any $w \in V_{pq}$ will lead M from p to q regardless of the stack content at p, and leave the stack at q in the same condition as it was at p. Different w's $\in L(V_{pq})$ may do this in different ways, which leads to different rules of G:

1) The stack may be empty only in p and in q, never in between. If so, w = a v b, for some $a, b \in A$, $v \in A^*$. And M includes the transitions (p, a, ε) -> (r, t) and (s,b, t) -> (q, ε). Add the rules: $V_{pq} \to a \ V_{rs} \ b$

2) The stack may be empty at some point between p and in q, in state r.
For each triple p, q, r $\in$ Q, add the rules: V_{pq} -> V_{pr} V_{rq}.
3) For each p $\in$ Q, add the rule V_{pp} -> ε .

The figure at left illustrates Rule1, at right Rule 2. If M includes the transitions (p, a, ε) -> (r, t) and (s,b, t) -> (q, ε), then one way to lead M from p to q with identical stack content at the start and the end of the journey is to break the trip into three successive parts: 1) to read a symbol 'a' and push 't'; 2) travel from r to s with identical stack content at the start and the end of this sub-journey; 3) to read a symbol 'b' and pop 't'.





5.3 Normal forms

When trying to prove that all objects in some class C have a given property P, it is often useful to first prove that each object O in C can be transformed to some equivalent object O' in some subclass C' of C. Here, 'equivalent' implies that the transformation preserves the property P of interest. Thereafter, the argument can be limited to the the subclass C', taking advantage of any additional properties this subclass may have.

Any CFG can be transformed into a number of "normal forms" (NF) that are (almost!) equivalent. Here, 'equivalent' means that the two grammars define the same language, and the proviso "almost" is necessary because these normal forms cannot generate the null string.

Chomsky normal form (right-hand sides are short):
All rules are of the form X -> Y Z or X -> a, for some non-terminals X, Y, Z $\in$ V and terminal a $\in$ A

Thm: Every CFG G can be transformed into a Chomsky NF G' such that L(G') = L(G) - {ε}.
Pf idea: repeatedly replace a rule X -> v w, $|v| \geq 1$, $|w| \geq 2$ by X -> Y Z, Y -> v, Z -> w, where Y and Z are new non-terminals used only in these new rules. Both right hand sides v and w are shorter than the original right hand side v w.
The Chomsky NF changes the syntactic structure of L(G), an undesirable side effect in practice. But Chomsky NF turns all syntactic structures into binary trees, a useful technical device that we exploit in later sections on the Pumping Lemma and the CYK parsing algorithm.

Greibach normal form (at every step, produce 1 terminal symbol at the far left - useful for parsing):
All rules are of the form X -> a w, for some terminal a $\in$ A, and some w $\in$ V*

Thm: Every CFG G can be transformed into a Greibach NF G' such that L(G') = L(G) - {ε}.
Pf idea: for a rule X -> Y w, ask whether Y can ever produce a terminal at the far left, i.e. Y ->* a v. If so, replace X -> Y w by rules such as X -> a v w. If not, X -> Y w can be omitted, as it will never lead to a terminating derivation.

5.4 The pumping lemma for CFLs

Recall the pumping lemma for regular languages, a mathematically precise statement of the intuitive notion "a FSM can count at most up to some constant n". It says that for any regular language L, any sufficiently long word w in L can be split into 3 parts, w = x y z, such that all strings x y^k z, for any $k \geq 0$, are also in L.

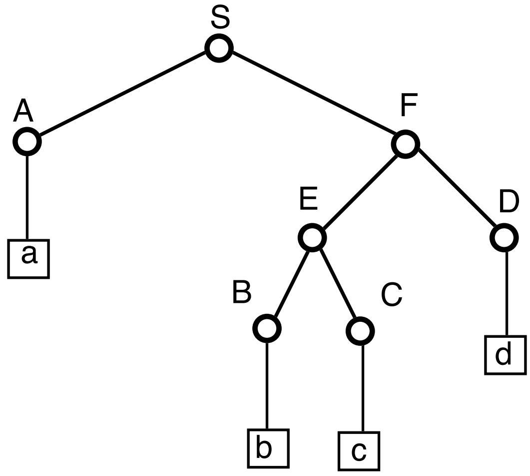
PDAs, which correspond to CFGs, can count arbitrarily high - though essentially in unary notation, i.e. by storing k symbols to represent the number k. But the LIFO access limitation implies that the stack can only be used to represent one single independent counter at a time. To understand what 'independent' means, consider a

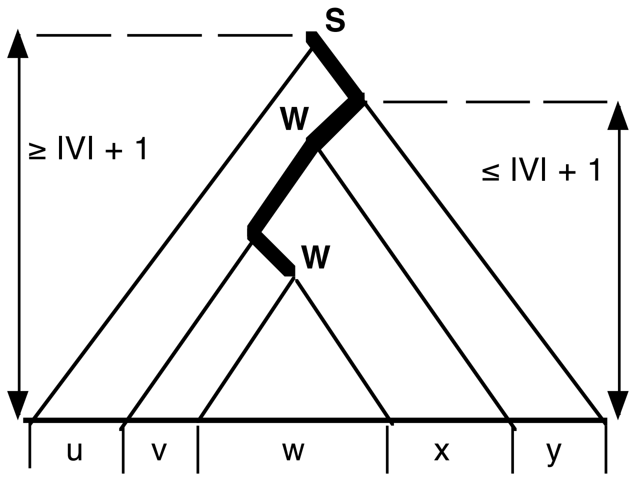
PDA that recognizes a language of balanced parenthesis expressions, such as ((([[..]]))). This task clearly calls for an arbitrary number of counters to be stored at the same time, each one dedicated to counting his own subexpression. In the example above, the counter for '(((' must be saved when the counter for '[[' is activated. Fortunately, balanced parentheses are nested in such a way that changing from one counter to another matches the LIFO access pattern of a stack - when a counter, run down to 0, is no longer needed, the next counter on top of the stack is exactly the next one to be activated. Thus, the many counters coded into the stack interact in a controlled manner, they are not independent.

The pumping lemma for CFLs is a precise statement of this limitation. It asserts that every long word in L serves as a seed that generates an infinity of related words that are also in L.

Thm: For every CFL L there is a constant n such that every $z \in L$ of length $|z| \geq n$ can be written as $z = u\ v\ w\ x\ y$ such that the following holds:
1) $v\ x \neq \varepsilon$, 2) $|v\ w\ x| \leq n$, and 3) $\mathbf{u\ v^k\ w\ x^k\ y \in L}$ **for all** $\mathbf{k \geq 0}$**.**

Pf: Given CFL L, choose any G = G(L) in Chomsky NF. This implies that the parse tree of any $z \in L$ is a binary tree, as shown in the figure below at left. The length n of the string at the leaves and the height h of a binary tree are related by $h \geq \log n$, i.e. a long string requires a tall parse tree. By choosing the critical length $\mathbf{n = 2^{|V| + 1}}$ we force the height of the parse trees considered to be $\mathbf{h \geq |V| + 1}$. On a root-to-leaf path of length $\geq |V| + 1$ we encounter at least $|V| + 1$ nodes labeled by non-terminals. Since G has only $|V|$ distinct non-terminals, this implies that on some long root-to-leaf path we must encounter 2 nodes labeled with the same non-terminal, say W, as shown at right.





For two such occurrences of W (in particular, the two lowest ones), and for some $u, v, y, x, w \in A^*$, we have: S ->* u W y, W ->* v W x and W ->* w. But then we also have W ->* v^2 W x^2, and in general, W ->* v^k W x^k, and S ->* u v^k W x^k y and S ->* u v^k w x^k y for all $k \geq 0$, **QED**.

For problems where intuition tells us"a PDA can't do that", the pumping lemma is often the perfect tool needed to prove rigorously that a language is not CF. For example, intuition suggests that neither of the languages $L1 = \{ 0^k\ 1^k\ 2^k / k \geq 0 \}$ or $L2 = \{ w\ w / w \in \{0, 1\} \}$ is recognizable by some PDA.

For L1, a PDA would have to count up the 0s, then count down the 1s to make sure there are equally many 0s and 1s. Thereafter, the counters is zero, and although we can count the 2s, can't compare that number to the number of 0s, or of 1s, an information that is now lost.

For L2, a PDA would have to store the first half of the input, namely w, and compare that to the second half to verify that the latter is also w. Whereas this worked trivially for palindromes, $w\ w^{\text{reversed}}$, the order w w is the worst case possible for LIFO access: although the stack contains all the information needed, we can't extract the info we need at the time we need it. The pumping lemma confirms these intuitive judgements.

Ex 1: L1 = $\mathbf{\{ 0^k\ 1^k\ 2^k / k \geq 0 \}}$ **is not context free.**

Pf (by contradiction): Assume L is CF, let n be the constant asserted by the pumping lemma.
Consider $z = 0^n 1^n 2^n = u\ v\ w\ x\ y$. Although we don't know where vwx is positioned within z, the assertion $|v\ w\ x| \le n$ implies that v w x contains at most two distinct letters among 0, 1, 2. In other words, one or two of the three letters 0, 1, 2 is missing in vwx. Now consider $u\ v^2\ w\ x^2\ y$. By the pumping lemma, it must be in L. The assertion $|v\ x| \ge 1$ implies that $u\ v^2\ w\ x^2\ y$ is longer than u v w x y. But u v w x y had an equal number of 0s, 1s, and 2s, whereas $u\ v^2\ w\ x^2\ y$ cannot, since only one or two of the three distinct symbols increased in number. This contradiction proves the thm.

Ex 2: L2 = { w w / w ∈ {0, 1} } is not context free.

Pf (by contradiction): Assume L is CF, let n be the constant asserted by the pumping lemma.
Consider $z = 0^{n+1}\ 1^{n+1}\ 0^{n+1}\ 1^{n+1} = u\ v\ w\ x\ y$. Using k = 0, the lemma asserts $z_0 = u\ w\ y \in L$, but we show that z_0 cannot have the form t t, for any string t, and thus that $z_0 \notin L$, leading to a contradiction. Recall that $|v\ w\ x| \le n$, and thus, when we delete v and x, we delete symbols that are within a distance of at most n from each other. By analyzing three cases we show that, under this restriction, it is impossible to delete symbols in such a way as to retain the property that the shortened string $z_0 = u\ w\ x$ has the form t t. We illustrate this using the example n = 3, but the argument holds for any n.
Given z = 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1, slide a window of length n = 3 across z, and delete any characters you want from within the window. Observe that the blocks of 0s and of 1s within z are so long that the truncated z, call it z', still has the form "0s 1s 0s 1s". This implies that if z' can be written as z' = t t, then t must have the form t = "0s 1s". Checking the three cases: the window of length 3 lies entirely within the left half of z; the window straddles the center of z; and the window lies entirely within the right half of z, we observe that in none of these cases z' has the form z' = t t, and thus that $z_0 = u\ w\ y \notin L$. QED

5.5 Closure properties of the class of CFLs

Thm (CFL closure properties): The class of CFLs over an alphabet A is closed under the regular operations union, catenation, and Kleene star.

Pf: Given CFLs L, L' ⊆ A*, consider any grammars G, G' that generate L and L', respectively. Combine G and G' appropriately to obtain grammars for $L \cup L'$, L L', and L*. E.g, if G = (V, A, P, S), we obtain
$G(L^*) = (V \cup \{S_0\}, A, P \cup \{ S_0 \to S\ S_0 ,\ S_0 \to \varepsilon \}, S_0)$.

The proof above is analogous to the proof of closure of the class or regular languages under union, catenation, and Kleene star. There we combined two FAs into a single one using series, parallel, and loop combinations of FAs. But beyond the three regular operations, the analogy stops. For regular languages, we proved closure under complement by appealing to **deterministic** FAs as acceptors. For these, changing all accepting states to non-accepting, and vice versa, yields the complement of the language accepted. This reasoning fails for CFL's, because deterministic PDAs accept only a subclass of CFLs. For non-deterministic PDAs, changing accepting states to non-accepting, and vice versa, does not produce the complement of the language accepted. Indeed, closure under complement does not hold for CFLs.

Thm: The class of CFLs over an alphabet A
is **not** closed under intersection and is not closed under omplement.

We prove this theorem in two ways: first, by exhibiting two CFLs whose intersection is provably not CF, and second, by exhibiting a CFL whose complement is provably not CF.

Pf ∩: Consider CFLs $L_0 = \{ 0^m 1^m 2^n \mid m, n \ge 1 \}$ and $L_1 = \{ 0^m 1^n 2^n \mid m, n \ge 1 \}$.
$L_0 \cap L_1 = \{ 0^k 1^k 2^k \mid k \ge 1 \}$ is **not** CF, as we proved in the previous section using the pumping lemma.

This implies that the class of CFLs is not closed under complement. If it were, it would also be closed under intersection, because of the identity: $L \cap L' = \neg(\neg L \cup \neg L')$. But we also prove this result in a direct way by exhibiting a CFL L whose complement is not context free. L's complement is the notorious language L2 =

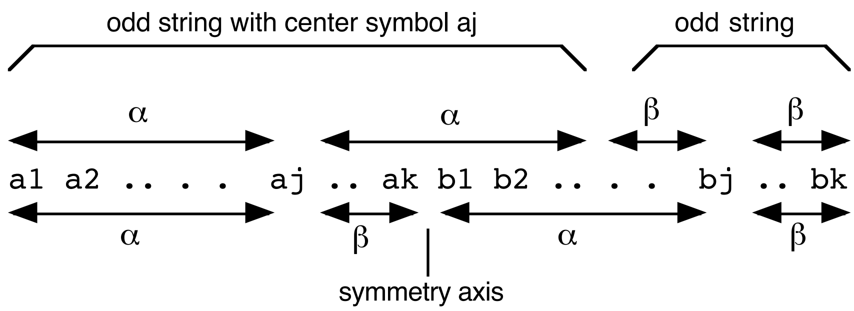
{ w w / w ∈ {0, 1} } , which we have proven not context free using the pumping lemma.

Pf ¬: We show that L = { u | u is **not** of the form u = w w } is context free by exhibiting a CFG for L:

S -> Y | Z| Y Z | Z Y
Y -> 1 | 0 Y 0 | 0 Y 1 | 1 Y 0 | 1 Y 1
Z -> 0 | 0 Z 0 | 0 Z 1 | 1 Z 0 | 1 Z 1

The productions for Y generate all odd strings, i.e. strings of odd length, with a 1 as its center symbol. Analogously, Z generates all odd strings with a 0 as its center symbol. Odd strings are not of the form u = w w, hence they are included in L by the productions S -> Y | Z . Now we show that the strings u of even length that are **not** of the form u = w w are precisely those of the form Y Z or Z Y.
First, consider a word of the form Y Z, such as the catenation of y = 1 1 0 **1** 0 0 0 and z = 1 **0** 1, where the center **1** of y and the center **0** of z are highlighted. Writing y z = 1 1 0 **1** 0 0 0 1 **0** 1 as the catenation of two strings of equal length, namely 1 1 0 **1** 0 and 0 0 1 **0** 1, shows that the former center symbols **1** of y and **0** of z have both become the 4-th symbol in their respective strings of length 5. Thus, they are a witness pair whose clash shows that y z ≠ w w for any w. This, and the analogous case for Z Y, show that the set of strings of the form Y Z or Z Y are in L.

Conversely, consider any even word u = a1 a2 .. aj .. ak b1 b2 .. bj .. bk which is **not** of the form u = w w. There exists an index j where aj ≠ bj, and we can take each of aj and bj as center symbol of its own odd string. The following example shows a clashing pair at index j = 4: u = 1 1 0 **0** 1 1 1 0 **1** 1.
Now u = 1 1 0 **0** 1 1 1 0 **1** 1 can be written as u = z y, where z = 1 1 0 **0** 1 1 1 ∈ Z and y = 0 **1** 1 ∈ Y.
The following figure how the various string lengths labeled α and β add up.

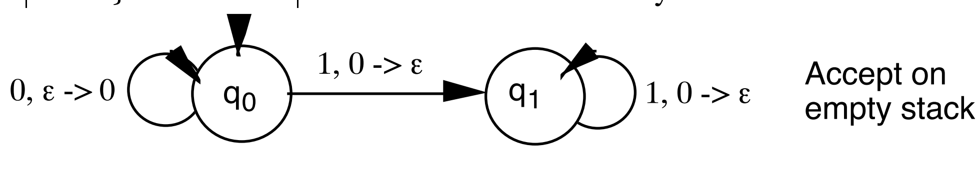


5.6 The "word problem". CFL parsing in time $O(n^3)$ by means of dynamic programming

Informally, the word problem asks: given G and w ∈ A*, decide whether w ∈ L(G).
More precisely: is there an algorithm that applies to any grammar G in some given class of grammars, and any w ∈ A*, to decide whether w ∈ L(G)?

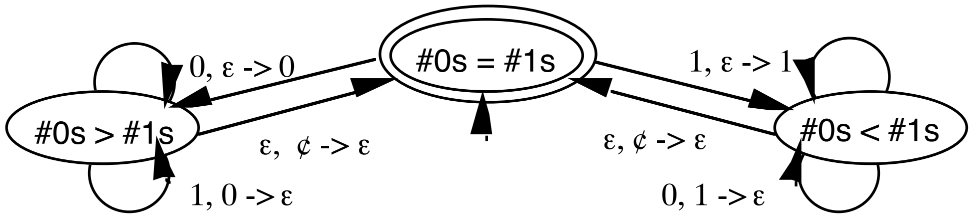
Many algorithms solve the word problem for CFGs, e.g: a) convert G to Greibach NF and enumerate all derivations of length ≤ |w| to see whether any of them generates w; or b) construct an NPDA M that accepts L(G), and feed w into M.

Ex1: L = { $0^k 1^k$ | k ≥ 1 }. G: S -> 01 | 0 S1. Use "0" as a stack symbol to count the number of 0s.



Ex2: L = {w ∈ {0, 1}* | #0s = #1s }. G: S -> ε | 0 Y' | 1 Z', Y' -> 1 S | 0 Y' Y', Z' -> 0 S | 1 Z' Z'
Invariant: Y' generates any string with an extra 1, Z' generates any string with an extra 0.
The production Z' -> 0 S | 1 Z' Z' means that Z' has two ways to meet its goal: either produce a 0 now and

follow up with a string in S, i.e with an equal number of 0s and 1s; or produce a 1 but create two new tasks Z'.



For CFGs there is a "bottom up" algorithm (Cocke, Younger, Kasami) that systematically computes all possible parse trees of all contiguous substrings of the string w to be parsed, and works in time $O(|w|^3)$. We illustrate the idea of the CYK algorithm using the following example:

Ex2a: L = $\{w \in \{0, 1\}^+ \mid \#0s = \#1s\}$. G: S -> 0 Y' | 1 Z', Y' -> 1S | 0 Y' Y', Z' -> 0 S | 1 Z' Z'

We exclude the nullstring in order to convert G to Chomsky NF. For the sake of formality, introduce Y that generates a single 1, similarly for Z and 0. Shorten the right hand side 0 Z' Z' by introducing a non terminal Z" -> Z' Z', and similarly Y" -> Y'Y'. Every w ∈ Z" can be written as w = u v, u ∈ Z', v ∈ Z'. As we read w from left to write, there comes an index k where #1s = #0s +1, and that prefix of w can be taken as u. The remainder v has again #1s = #0s +1.

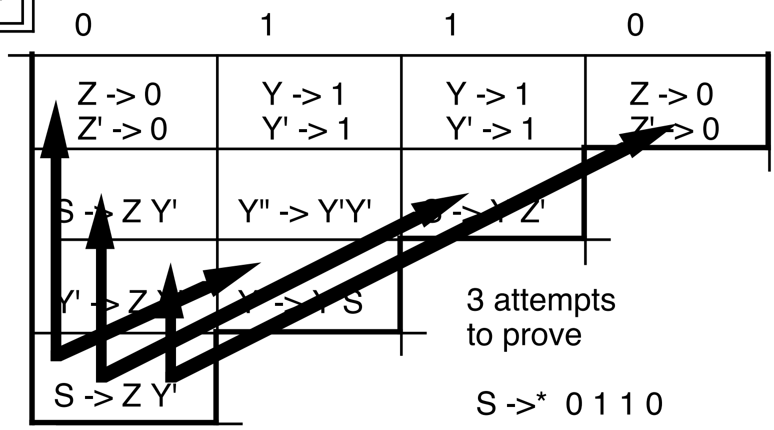
The grammar below maintains the invariants: Y generates a single "1"; Y' generates any string with an extra "1"; Y" generates any string with 2 extra "1". Analogously for Z, Z', Z" and "0".

S -> Z Y' I Y Z'	start with a 0 and remember to generate an extra 1, or start with a 1 and ...
Z -> 0, Y -> 1	Z and Y are mere formalities
Z' -> 0 I Z S I Y Z"	produce an extra 0 now, or produce a 1 and remember to generate 2 extra 0s
Y' -> 1 I Y S I Z Y"	produce an extra 1 now, or produce a 0 and remember to generate 2 extra 1s
Z" -> Z' Z', Y" -> Y'Y'	split the job of generating 2 extra 0s or 2 extra 1s

The following table parses a word w = 001101 with |w| = n. Each of the n (n+1)/2 entries corresponds to a substring of w. Entry (L, i) records all the parse trees of the substring of length L that begins at index i. The entries for L = 1 correspond to rules that produce a single terminal, the other entries to rules that produce 2 non-terminals.

L \ w =	0	0	1	1	0	1
1	Z -> 0 Z' -> 0	Z -> 0 Z' -> 0	Y -> 1 Y' -> 1	Y -> 1 Y' -> 1	Z -> 0 Z' -> 0	Y -> 1 Y' -> 1
2	Z'' -> Z' Z'	S -> Z Y'	Y'' -> Y'Y'	S -> Y Z'	S -> ZY'	
3	Z' -> Z S	Y' -> Z Y''	Y' -> Y S	Y' -> Y S		
4	S -> Z Y'	S -> Z Y'	$\boxed{\text{Y'' -> Y'Y'}}$			
5	Z'' -> Z S	Y' -> Z Y''				
6	S -> Z Y'					

Notice the framed rule
Y'' -> Y' Y'
matches in 2 distinct ways
(ambiguous grammar)



The picture at the lower right shows that for each entry at level L, we must try (L-1) distinct ways of splitting that entry’s substring into 2 parts. Since (L-1) < n and there are n (n+1)/2 entries to compute, the CYK parser works in time $O(n^3)$.

Useful CFLs, such as parts of programming languages, should be designed so as to admit more efficient parsers, preferably parsers that work in linear time. LR(k) grammars and languages are a subset of CFGs and CFLs that can be parsed in a single scan from left to right, with a look-ahead of k symbols.

5.7 Context sensitive grammars and languages

The rewriting rules B -> w of a CFG imply that a non-terminal B can be replaced by a word $w \in (V \cup A)^*$ "in any context". In contrast, a context sensitive grammar (CSG) has rules of the form:
u B v -> u w v, where $u, v, w \in (V \cup A)^*$,
implying that B can be replaced by w only in the context "u on the left, v on the right".

It turns out that this definition is equivalent (apart from the nullstring ε) to requiring that any CSG rule be of the form v -> w, where $v, w \in (V \cup A)^*$, and $|v| \le |w|$. This monotonicity property (in any derivation, the current string never gets shorter) implies that the word problem for CSLs: "given CSG G and given w, is $w \in L(G)$?" is decidable. An exhaustive enumeration of all derivations up to the length |w| settles the issue.

As an example of the greater power of CSGs over CFGs, recall that we used the pumping lemma to prove that the language $0^k 1^k 2^k$ is not CF. By way of contrast, we prove:
Thm: $L = \{ 0^k 1^k 2^k / k \ge 1 \}$ is context sensitive.

The following CSG generates L. Function of the non-terminals V = {S, B, C, Y, Z}: each Y and Z generates a 1 or a 0 at the proper time; B initially marks the beginning (left end) of the string, and later converts the Zs into 0s; C is a counter that ensures an equal number of 0s, 1s, 2s are generated. Non-terminals play a similar role as markers in Markov algorithms. Whereas the latter have a deterministic control structure, grammars are non-deterministic.

S -> B K 2	at the last step in any derivation, B K generates 01, balancing this '2'
K -> Z Y K 2	counter K generates $(ZY)_k\ 2_k$
K -> C	when k has been fixed, C may start converting Ys into 1s
Y Z -> Z Y	Zs may move towards the left, Ys towards the right at any time
B Z -> 0 B	B may convert a Z into a 0 and shift it left at any time
Y C -> C 1	C may convert a Y into a 1 and shift it right at any time
B C -> 01	when B and C meet, all permutations, shifts and conversions have been done

Hw 5.1: Context-free grammars and pushdown automata

Consider the context-free grammar G with non-terminals S and P, start symbol S, terminals '(', ')' and '0', and productions: S -> S P | ε ; P -> (S) | 0 .

a) Draw a derivation tree for each of the 4 shortest strings produced by G.
b) Prove or disprove: the grammar G is unambiguous.
c) Design a pushdown automaton M to accept the language L(G). Let M be deterministic if possible, or non-deterministic if necessary.

Ex: Show that $L = \{ww \mid w \in \{0, 1\}^* \}$ is context sensitive.

Ex: Recall the grammar G1 of arithmetic expressions, e.g. in the simplified form:

E -> T | EAT, T -> F | TMF, F -> N | V | (E), A -> + | - , M -> * | /

For simplicity's sake, we limit numbers to single bits, i.e. N -> 0 | 1, and use only 3 variables, V -> x | y | z

a) Extend G1 to a grammar G2 that includes function terms, such as f(x) and g(1 - x/y)
Use only 3 function symbols defined in a new production H -> f | g | h

b) Extend G2 to a grammar G3 that includes integration terms, such as S [a, b] f(x) dx, a linearized form of "integral from a to b of f(x) dx".

c) Discuss the strengths and weaknesses of CFGs as tools to solve the tasks a) and b).

Ex: Let $L = \{ww \mid w \in \{0, 1\}^* \}$
a) Prove that L is **not** context free, and b) prove that L is context sensitive.

End of Ch 5